实验二：语义分析

组长：周心萌 141220161 ddzhouxm@163.com
组员：银琦 141220132 141220132@smail.nju.edu.cn

1. 实现功能
   a) 功能介绍
      编写一个对于 c—代码进行语义分析的程序，分析多种语义错误并进行输出
   b) 实现过程
      i. 定义结构
      对于符号表我们采用了闭散列的数据结构，主要结构成员变量有变量名和存放变量信息的结点，变量信息又有变量类型，变量层次（要求 2.2），以及函数或者类型的具体信息，在存储类型具体信息的时候，由于函数的变量以及普通的变量都用该结构存储，所以采用了 union 的形式，节省程序空间。对于结构体的存放，我们采用了链表的形式，每个结构体有结构体名称，以及结构体的变量信息，结构体变量同样采用链表形式，每一个节点存放变量的类型以及名称。
      结构体：

```
struct Structure{
    char name[32];
//  struct array *S_array;
//  struct structField *f;
    struct Structure *next;
    int depth;
    struct argList *sMember;
    int inStruct;
//  int
};
```

      符号表

```
union msgSymbol{
    struct msgVar *Vmsg;
    struct msgFun *Fmsg;
};

struct NodeSymbol{
    int FunOrVar;//fun 0 var 1
    struct NodeSymbol *next;
    int line;
    int depth;
    union msgSymbol Smsg;
};

struct symbol{
    struct NodeSymbol *node;
    char name[32];
    int ifUse;//0 not use 1 use
};
```

      ii. 语义分析
      定义三个符号表，分别用于存储普通类型（int、float、array、struct）变量，存储函数类型以及程序中定义的结构体。从语法树的根节点开始，根据 c—中的文法规则对每一个语法树的节点进行对应类型的语义分析，

```
extern void init();
extern void startSemantic();
extern void isProgram(struct treeNode *node);
extern void isExtDefList(struct treeNode *node);
extern void isExtDef(struct treeNode *node);
extern void isExtDecList(struct treeNode *node,struct Structure *s, enum type MyType);

extern void isSpecifier(struct treeNode *node,enum type *MyType, struct Structure *s);
extern void isStructSpecifier(struct treeNode *node,enum type *MyType, struct Structure
extern void isOptTag(struct treeNode *node, struct Structure *s);
extern void isTag(struct treeNode *node,struct Structure *s);
```

      iii. 要求 2.2
      我们采用实验讲义中所提及的 Imperative Style 风格。使用一个全局变量 depth 来指示当前程序所处的层次，每当一个 compst（{}语句块）开始时，depth++，结束时遍历变量符号表，将所有仅处于当前层次的变量节点删除，同样。在创建变量时，如果一个变量与需要创建的变量同名但所处层次不同，则不报变量

名重复错

Compst 结束时操作

```
for(i=0;i<1000;i++){
    if(varSymbol[i]!=NULL){
        if(varSymbol[i]->node->depth == depth){
            varSymbol[i]->node = varSymbol[i]->node->next;
            if(varSymbol[i]->node == NULL)
                varSymbol[i] = NULL;
        }
    }
}
depth--;
```

c) 完成情况

在要求 2.2 下可以完成对测试文件中 17 种错误类型的判断。

2. 编译环境和输入格式

a) 编译环境

Ubuntu 12.04　flex 2.5.35　bison 2.5

b) 输入格式

进入 Code 文件夹，输入 make，对于在 Code 外 Test 文件夹中的测试文件，输入 make test；对于在 Code 内的测试文件， 输入./ parser filename。

3. 遇到的问题

a) 在完成要求 2.2 时发生段错误，经考察是在{}语句块结束时删除变量的节点仅仅删除了变量的 node 信息而未清空变量名以及对应的符号表信息，从而在定义新的变量时仍然找到了同名的变量。

b) 对于 args 语义分析时原本采用 void 的返回类型并且将 arglist 的指针类型当做 args 的形参，在 args 分析的时候对 arglist 进行修改，但在实际操作过程中发现在 args 中修改的 arglist 在 args 分析结束后未被修改或者修改后的信息不符合实际应该得到的信息，后改成采用返回值的方式返回 arglist 变量，即得到解决

c) 由于结构体中的变量也属于符号表中变量定义的范畴，导致错误 15 被误报为错误 3, 后再变量表中加入 instruct 指示变量解决该问题, 对于在结构体中的变量 instruct 设置为 1，不在结构体中的 instruct 设置为 0，判断时根据 instruct 的值来决定报错误 3 还是错误 15

d) 在实验过程中不够细致，出现许多小错误，比如未将节点的名字进行拷贝，导致在之后判断的时候读到了空 ;未对数组类型进行判断, 导致数组类型被误判为整型等。